Transformer-based model compression

July 19, 2023

Plan

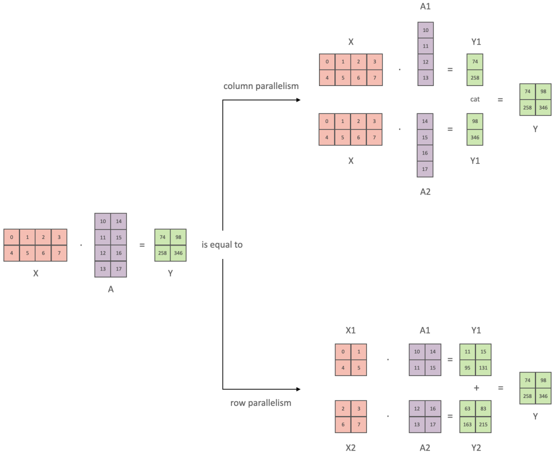
Training Parallelism

- **Data parrallelism** - pieces of a given batch are placed on a different GPU cards
- **Tensor parrallelism** - pieces of a model (blocks, layers, parts of the layers) are placed on a different GPU cards

Data Parallelism

- It creates and dispatches copies of the model, one copy per each accelerator.
- It shards the data to the n devices. If full batch has size B, now size is $\frac{B}{n}$.
- It finally aggregates all results together in the backpropagation step, so resulting gradient in module is average over n devices.

Tensor parrallelism



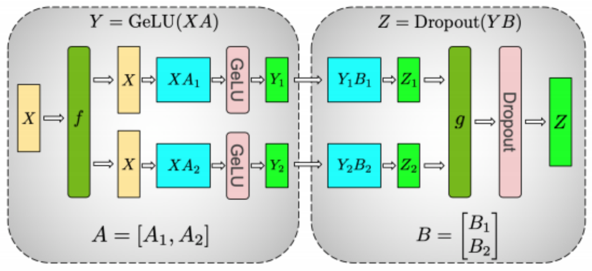
Different ways of splitting the matrix between several GPUs

Tensor parrallelism

A column-wise splitting provides matrix multiplications XA_1 through XA_n in parallel, then we will end up with N output vectors $Y_1, \ldots, Y_n$ which can be fed into GeLU independently

$$[Y_1, Y_2] = [GeLU(XA_1), GelU(XA_2)]$$

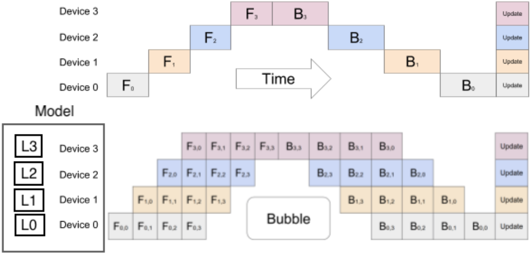
Using this principle, we can update an MLP of arbitrary depth, without the need for any synchronization between GPUs until the very end [1]:



(a) MLP

[1] Megatron

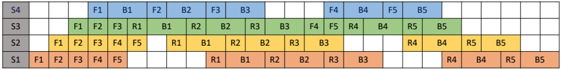
Pipelining



Top: The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one accelerator is active at a time. ***Bottom:*** *GPipe divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on separate micro-batches at the same time.*

Pipelining

Interleaved pipelining aims to reduce "bubble" size.



(a) Varuna Schedule

(b) Gpipe Schedule

Varuna [2] model scheduler. F - forward pass, B- backward pass, R - recomputation. Varuna recomputes activations by re-running the forward computation, sinse activations take lot of of memory (checkpointing).

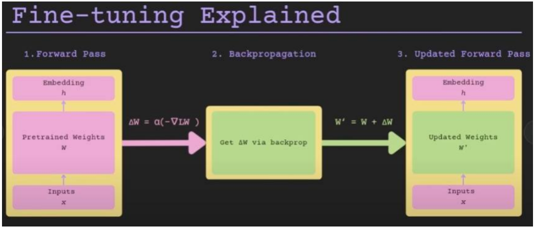
[2] https://arxiv.org/pdf/2111.04007.pdf

Matrix decompositions

- Singular Value Decomposition (SVD)
- Kronecker Decomposition

LORA:Low-rank adaptation of large language models [3]

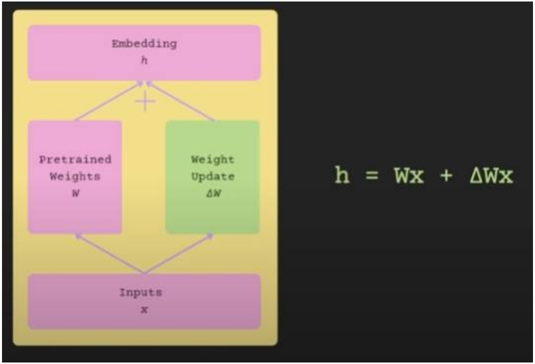
Fine-tuning: if the model has 100 billion trained parameters, storing all of them in memory becomes a significant bottleneck.



[3] https://arxiv.org/pdf/2106.09685.pdf

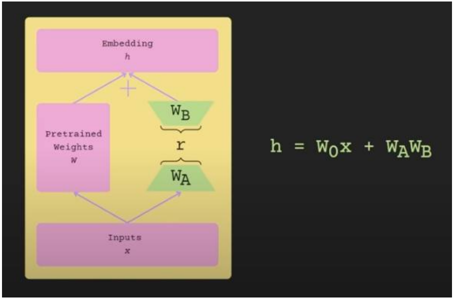
LORA:Low-rank adaptation of large language models

Thus, the pre-trained weights remain static and we only manipulate the delta weights. This is a crucial aspect of the algorithm.



LORA:Low-rank adaptation of large language models

The low-rank approximation is generated by using a technique called singular value decomposition (SVD). SVD decomposes the base model into a set of rank-1 matrices. These rank-1 matrices are then combined to form the target model.



LORA

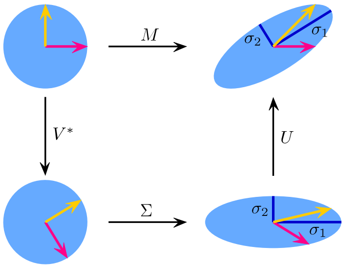
Results on GLUE (Roberta)

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{D})*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} (Adpt^{D})*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**

Singular Value Decomposition (SVD)

- The Singular Value Decomposition (SVD) of a matrix $\mathbf{M} \in \mathcal{R}^{m \times n}$ is a factorization $\mathbf{M} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^*$ where $\mathbf{U} \in \mathcal{R}^{m \times m}$, $\boldsymbol{\Sigma} \in \mathcal{R}^{m \times n}$ - is a diagonal matrix, $\mathbf{V} \in \mathcal{R}^{n \times n}$.
- $\tilde{\mathbf{M}}$ is a **truncated approximation M** if $\tilde{\mathbf{M}} = \mathbf{U}\tilde{\Sigma}\mathbf{V}^*$, where $\tilde{\boldsymbol{\Sigma}}$ is a $\boldsymbol{\Sigma}$ a non-zero only the r largest singular values.

Singular Value Decomposition (SVD)



$$M = U \cdot \Sigma \cdot V^*$$

Figure: The interpretation of SVD. The operator of transformation **M** is decomposed into rotation **U**, scaling $\mathbf{\Sigma}$ and rotation back $\mathbf{V^T}$.

Three factor matrices can be treated as rotation, scaling and rotation back operations. We translate **M** into new space, truncate the elements with minot weights, and then transfer the Matrix back.

Kronecker product

Kronecker product

For an $B \in \mathbb{R}^{I \times J}$ matrix a $C \in \mathbb{R}^{K \times L}$, the standard (Right) Kronecker product, $B \otimes C$ is the $\mathbb{R}^{IK \times JL}$ matrix

$$\left[\begin{array}{cc|cc} b_{1,1}c_{1,1} & b_{1,1}c_{2,1} & b_{2,1}c_{1,1} & b_{2,1}c_{2,1} \\ b_{1,1}c_{1,2} & b_{1,1}c_{2,2} & b_{2,1}c_{1,2} & b_{2,1}c_{2,2} \\ \hline b_{1,2}c_{1,1} & b_{1,2}c_{2,1} & b_{2,2}c_{1,1} & b_{2,2}c_{2,1} \\ b_{1,2}c_{1,2} & b_{1,2}c_{2,2} & b_{2,2}c_{1,2} & b_{2,2}c_{2,2} \end{array}\right] = \left[\begin{array}{cc} b_{1,1} & b_{2,1} \\ b_{1,2} & b_{2,2} \end{array}\right] \otimes \left[\begin{array}{cc} c_{1,1} & c_{2,1} \\ c_{1,2} & c_{2,2} \end{array}\right]$$

Tensors decompositions

- Notation
- Canonical Plyadic
- Tucker decomposition
- Tensor Train decomposition
- Tensor Train Matrix Decomposition

Notation

Vectorization

Isomorphism that maps the element of tensor to vector:

$\mathbb{R}^{I_1 \times \cdots \times I_N} \to (I_1 \times \cdots \times I_N)$

$$j = \sum_{k=0}^{N} i_k \prod_{m=k+1}^{N} I_m = i_N + i_{N-1} \cdot I_N + i_{N-2} \cdot I_N I_{N-2} + \cdots + i_1 \cdot I_2 \ldots I_N$$

or

$$j = \sum_{k=0}^{N} \left[i_k \prod_{m=0}^{k-1} I_m \right] = i_1 + i_2 I_1 + i_3 I_1 I_2 + i_N I_1 \ldots I_N$$

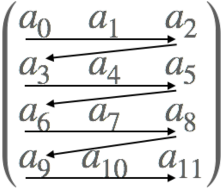


Figure: Numpy, Tensorly

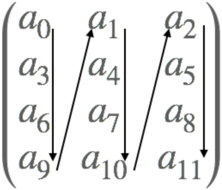


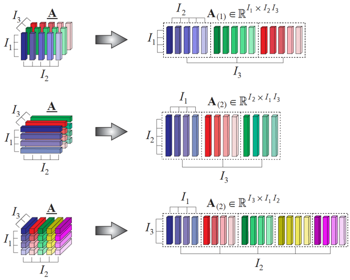
Figure: Matlab tools

Notation

Unfolding

Tensor unfolding, or matrization, is a fundamental operation and a building block for most tensor methods. Considering a tensor as a multi-dimensional array, unfolding it consists of reading its element in such a way as to obtain a matrix instead of a tensor.

$\hat{X} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N} \rightarrow X_{(n)} \in \mathbb{R}^{I_n \times I_1 \cdot \cdots \cdot I_N}$



Mode-1, mode-2, and mode-3 matricizations of a 3rd-order tensor

Notation

Outer product

Outer product of tensors N-order tensor $\circ$ $\hat{A} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_N}$ and M-order tensor $\hat{A} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_M}$ is a (N+M) order tensor $\hat{C}$:

$$c_{i_1 \dots i_N j_1 \dots j_M} = a_{i_1 \dots i_N} b_{j_1 \dots j_M} \tag{1}$$

Mode-n product

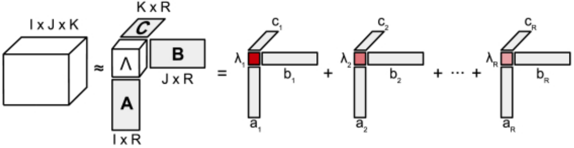
Mode-n product $\times_n$ of tensor $\hat{A} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \cdots \times I_n \times I_N}$ and matrix $B \in \mathbb{R}^{J \times I_n}$ tensor $\hat{C} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \cdots \times J \cdots \times I_N}$, that has elements:

$$c_{i_1 \dots i_{n-1} j i_{n+1} \dots j_N} = \sum_{i_n} a_{i_1 \dots i_n \dots i_N} b_{j i_n} \tag{2}$$

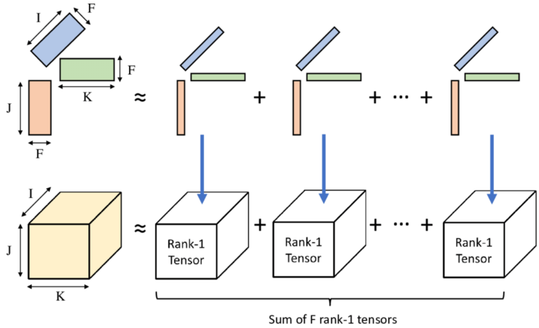
CP decomposition

In common, tensor decomposition is a scheme for representing a multidimensional object as a sequence of elementary operations acting on other objects with smaller sizes or dimensions. The Canonical Polyadic tensor decomposition (CPD) introduses a N order tensor $X \in \mathcal{R}^{I_1, I_2, \dots I_N}$ as a sum of $R < N$ tensors of rank 1, also known as Kruskal tensors:

$$X \approx \sum_{r=1}^{F} \lambda_r b_r^{(1)} \circ b_r^{(2)} \cdots \circ b_r^{(n)} = \Lambda \times_1 \mathbf{B}^{(1)} \times_2 \mathbf{B}^{(2)} \cdots \times_N \mathbf{B}^{(N)} \qquad (3)$$



CP decomposition



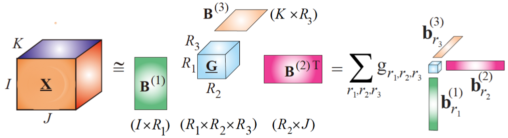
Example:

- $\underline{X} \in [100, 200, 300]$
- rank $= 50$
- $A \in [100, 50], B \in [200, 50], C \in [300, 50]$

Tucker decomposition

$$X \approx \sum_{r_1=1}^{R_1} \cdots \sum_{r_N=1}^{R_N} g_{r_1, r_2 .. r_N} b_{r_1}^{(1)} \circ b_{r_2}^{(2)} \cdots \circ b_{r_n}^{(n)} = \underline{\mathbf{G}} \times_1 \mathbf{B}^{(1)} \times_2 \mathbf{B}^{(2)} \cdots \times_N \mathbf{B}^{(N)} \tag{4}$$

Scheme of the Tucker decomposition



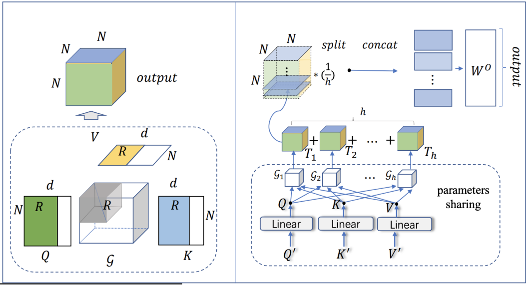
Example: $\underline{X} \in [100, 200, 300]$, rank $= 50$, $\underline{G} \in [50, 50, 50]$,
$A \in [100, 50], B \in [200, 50], C \in [300, 50]$

CPD vs. Tucker

Decomp Features	CPD	Tucker
Convergency	Worse	Better, more extensive set of possible elements in approximation
Parameter in the compression variant	$O(NI_{max}R)$	$O(R^N + O(NI_{max}R))$

Factorization: Attention block [4]

- Attention Block (sum of multi-head output) is represented as a Block-Term Tensor decomposition (right part ofthe figure).
- Block-Term Tensor decomposition is CP decomposition, where elements have a form of Tucker representations. Factor matrices are shared across CP elements, core tensors are different.
- Single attention is represented by Tucker decomposition (left part of figure).



[4] Attention factorization

Factorization: Attention block

More specifically, this :

$$\mathrm{Att}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

turns to this:

$$\mathrm{Att}_{\mathrm{TD}}(Q, K, V) = G \times_1 Q_F \times_2 K_F \times_3 V_F = \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N} G_{ijk} Q_{F\,i} \circ K_{F\,j} \circ V_{F\,k},$$

where Q_F, K_F, V_F have sizes $N \times r$ instead of $N \times d_k$

Table: Performance of Tensorized Transformer with and without compression of attention module

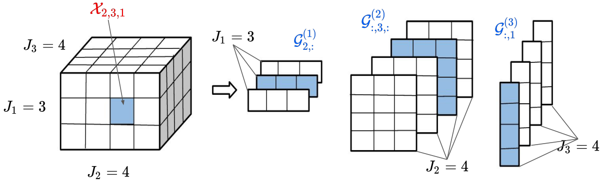
Language modelling			Translation		
Model	Parameters	PPL	Models	Parameters	BLEU
Transformer XL base	0.46 B	23.5	-	-	-
Transformer XL large	0.8 B	21.8	Transformer	51M	34.5
Tensorized core-1	0.16 B	20.5	Tensorized core-1	21M	34.1
Tensorized core-2	0.16 B	19.5	Tensorized core-2	21,2M	34.9

Tensor Train (TT) Format

Tensor Train (TT) decomposition represents a N - order object as a sequence of third-order tensors $G_1, \ldots, G_N$, adjacent to each other along one of the axes.

$$\mathcal{X}(i_1 \ldots i_N) = \underbrace{\mathcal{G}^{(1)}[i_1, :]}_{1 \times R_1} \underbrace{\mathcal{G}^{(2)}[:, i_2, :]}_{R_1 \times R_2} \ldots \underbrace{\mathcal{G}^{(N-1)}[:, i_{N-1}, :]}_{R_{N-2} \times R_{N-1}} \underbrace{\mathcal{G}^{(N)}[:, i_N]}_{R_{N-1} \times 1}$$

$\mathcal{G}_k \in \mathbb{R}^{R_k \times I_k \times R_{k+1}}$, $k = \overline{1, N}$, are 3-dimentional *core tensors (cores)* of TT decomposition. The $R_1, \ldots, R_k$ are called *TT-ranks*



Tensor Train

Algorithm 1 Full-to-TT compression algorithm.

Require: $n_1 \times n_2 \ldots \times n_d$ tensor $\mathbf{A}$, required accuracy ε.

Ensure: Cores $G_k, k = 1, \ldots, d$, of the TT-decomposition.

Unfoldings size: $N_l = n_1, N_r = \prod_{k=2}^{d} n_k$.
Temporary tensor: $\mathbf{B} = \mathbf{A}$.
First unfolding: $M = \mathrm{reshape}(\mathbf{B}, [N_l, N_r])$.
Compute truncated SVD: $M \approx U \Lambda V^\top$, and set r to be the approximate rank of M.
Set $G_1 := U$, $M := \Lambda V^\top, r_1 = r$.
{Process other modes}
for $k = 2$ to $d - 1$ **do**
  Set dimensions: $N_l := n_k, N_r := \frac{N_r}{n_k}$.
  Construct unfolding: $M := \mathrm{reshape}(M, [rN_l, N_r])$.
  Compute truncated SVD: $M \approx U \Lambda V^\top$, and set $r_k = r$ to be the approximate rank of M.
  Reshape and permute matrix U into a tensor:

$$G_k := \mathrm{reshape}(U, [r_{k-1}, n_k, r_k]), G_k := \mathrm{permute}(G_k, [2, 1, 3]).$$

  Recompute M: $M := \Lambda V^\top$.
end for
$G_d = M^\top$.

Represent matrix of the FC layer as a Tensor Train Matrix

$$\mathcal{T}(i_1, j_1, \ldots, i_M, j_M) \approx \sum_{r_1=1}^{R_1} \cdots \sum_{r_{M-1}=1}^{R_{M-1}} G^{(1)}(i_1, j_1, r_1) G^{(2)}(r_1, i_2, j_2, r_2) \ldots G^{(M)}(r_{M-1}, i_M, j_M)$$

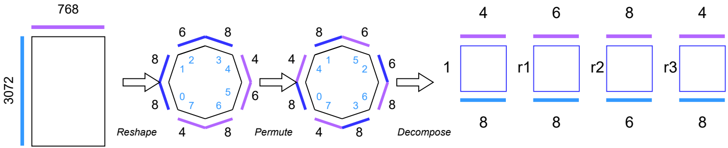
where $\mathcal{G}^{(m)} \in \mathbb{R}^{R_{m-1} \times I_m \times J_m \times R_m}$

- If matrix of FC layer $\mathbf{W} \in \mathbb{R}^{D_{in} \times D_{out}}$, where $D_{in} = \prod_{m=1}^{M} I_m$ and $D_{out} = \prod_{m=1}^{M} J_m$,
- Than compression rate (acoording to parameters number):

$$c\_rate = \frac{R(I_1 J_1 + I_M J_M) + R^2 \sum_{m=2}^{M-1} I_m J_m}{\prod_{m=1}^{M} I_m J_m}$$

Linear Layer as TTM

We reshape a layer weights matrix to an N-dimensional object and represent it as a TT product. The key idea is to create the maximum possible number of kernels of the minimum size - in this case, we can get the best compression.



Issue in Signal Propagation

If we represent the FC layer as a sequence of $\mathcal{G}$ cores, we assume Forward pass as a contraction process between input activations $\mathbf{X}$ and all these cores.

Opt-Einsum library generates a string-type expression, which:

1. defines the shapes of input and resulting tensors (e.g. "ikl,lkj-¿ij")
2. defines contraction schedule (e.g., firstly along axis 'l' and then along axis 'k').

The contraction between a set of multidimensional objects may not be memory-optimal.

Layer	TTM-16	Fully-Connected
Backprop Strategy	Torch Autodiff	Torch Autodiff
Single Layer, Batch 16	1100 MB	395 Mb

Issue in Signal Propagation

Forward Pass

- **Einsum** is a torch.opt.einsum, contaction scheduler optimized by the time
- **Custom Einsum** we set fixed contraction scheduler by ourself, multiplying the cores sequentially

Backward Pass

A gradient computation might be considered as a tensor contraction:

$$\frac{\partial \mathcal{L}}{\partial \mathcal{G}_m} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}} \frac{\partial \mathbf{W}}{\partial \mathcal{G}_m} = \mathbf{X}^T \frac{\partial \mathcal{L}}{\partial \mathcal{Y}} \frac{\partial \mathbf{W}}{\partial \mathcal{G}_m}.$$

- **Full Einsum** for all $\frac{\partial \mathcal{L}}{\partial \mathcal{G}_m}$ we can share intermediate results for contractions steps.
- **Full Matrix** tensors $\mathcal{X}$ and $\frac{\partial \mathcal{L}}{\partial \mathcal{Y}}$ are convolved along batch axis, and the rest schedule is further optimized

Optimization strategy in signal propagation

Optimization in Forward/Backward Strategies

Forward	Backward	Memory, Mb	Time, ms
Einsum	Torch Autodiff	1008	23.6
Einsum	Full Einsum	192	55.7
Einsum	**Full Matrix**	**192**	**17.5**
Custom Einsum	Torch Autodiff	2544	58.4
Custom Einsum	Full Einsum	192	84
Custom Einsum	Full Matrix	192	125

LA structures as PyTorch Layers

We can replace the weight matrix in Embedding, Attention or Linear layer with proposed algebraic structures: Truncated SVD, product TTM cores, and Kronecker products.

Way of the replacement

- Replace Layers in Transformer + Training Transformer from scratch:
- Replace Layers in Transformer by decomposing matrix in pre-trained Layer (usually is followed by significant quality drop) + fine-tune
- Replace Layers in Transformer + Training Transformer from scratch (i.e. distillation)

TTM Layers in BERT, Decomposition + Fine-Tuning

Method	C. Rate	AVG	STSB	CoLA	MNLI	MRCP	QNLI	QQP	RTE	SST2	WNLI
Full	109mln	0.79	0.88	0.57	0.84	0.9	0.91	0.87	0.67	0.92	0.54
SVD	53 mln	0.35	0.2	0	0.36	0.16	0.53	0.32	0.5	0.51	0.56
TTM		0.44	0.58	0.023	0.36	0.27	0.56	0.45	0.50	0.71	0.5
SVD	69 mln	0.45	0.61	0.03	0.36	0.16	0.51	0.62	0.47	0.73	0.56
TTM		0.44	0.65	0.01	0.40	0.15	0.53	0.51	0.47	0.73	0.47
SVD	102 mln	0.7	0.8	0.21	0.82	0.76	0.89	0.86	0.49	0.9	0.56
TTM		0.75	0.87	0.51	0.79	0.86	0.87	0.86	0.64	0.91	0.47

Table: The results of different types of compression of BERT for experiment with task-oriented fine-tining and further compression (Single-train).

Adding Fisher Information

Fisher matrix determines the weights importance during the task-specific model training.

$$I_w = E\left[\frac{\partial}{\partial \omega} logp(D|\omega)^2\right]$$

We inject the it into decomposition algorithms to minimize the gap between decomposition objective and task-oriented objective.
We inject the Fisher information into decomposition algorithms to minimize the gap between decomposition and task-oriented objectives

SVD

$$\hat{W} = \tilde{I}_w W = USV^T$$

$$\hat{U} = \tilde{I}_w^{-1} U$$

Adding Fisher Information

TTM

Algorithm 11 Fisher-Weighted TTM decomposition.

Input: Matrix of layer weights $\mathcal{W}$, matrix of Fisher weights $\mathcal{I}_{\mathcal{W}}$, shapes $I_1, J_1, \ldots, I_d, J_d$

Output: Cores $\mathcal{G}^k$, $k = 1 \ldots d$ of the TTM decomposition

$\mathcal{B} = \mathcal{W}.\text{reshape}(I_1, J_1, \ldots, I_d, J_d)$,

$\mathcal{B}_{\mathcal{I}} = \mathcal{I}_{\mathcal{W}}.\text{reshape}(I_1, J_1, \ldots, I_d, J_d)$,

$\mathcal{C} = \text{permute}(\mathcal{B})$, $\mathcal{C}_{\mathcal{I}} = \text{permute}(\mathcal{B}_{\mathcal{I}})$

for k in $\{1, \ldots, d-1\}$ **do**

  $N_l = n_k$

  $N_r = n_1 \ldots n_{k-1}, n_{k+1}, n_{d-1}$

  Unfolding $M = \mathcal{C}.reshape(N_1, rN_r)$,

  Unfolding $M_I = \mathcal{C}_{\mathcal{I}}.reshape(N_1, rN_r)$

  $\tilde{M}_I = diag(M_I)$

  $\tilde{M}_I M = USV^T$ truncated to r_k

  $\tilde{U} = \tilde{M}_I^{-1} U$, $M = SV^T$

  $G_k = \tilde{U}.\text{reshape}(r_k, n_k, r_{k+1})$

  $G_k = G_k.\text{permute}(2, 1, 3)$

end for

Adding Fisher Information

Method	C. Rate	AVG	STSB	CoLA	MNLI	MRCP	QNLI	QQP	RTE	SST2	WNLI
Full	109mln	0.79	0.88	0.57	0.84	0.9	0.91	0.87	0.67	0.92	0.54
FWSVD	3*53 mln	0.37	0.47	0	0.32	0.15	0.49	0.31	0.52	0.49	0.56
SVD		0.35	0.2	0	0.36	0.16	0.53	0.32	0.5	0.51	0.56
FWSVD	3*69 mln	0.51	0.40	0.06	0.49	0.46	0.63	0.78	0.56	0.70	0.54
SVD		0.45	0.61	0.03	0.36	0.16	0.51	0.62	0.47	0.73	0.56
FWSVD	3*102 mln	0.77	0.88	0.55	0.83	0.86	0.89	0.87	0.64	0.91	0.47
SVD		0.7	0.8	0.21	0.82	0.76	0.89	0.86	0.49	0.9	0.56

Table: Experiments for SVD compression in a BERT layers with and without Fisher information (Single-train).

TTM Layers in GPT-2, From Scratch

Experiments on a GPT-2 model of a small size

Model	Training	Validation	Number of parameters	%	Perplexity
GPT-2 small	Wikitext-103 train	Wikitext-103 test	124439808	100%	17.67
GPT-2 small TT rank 32	Wikitext-103 train	Wikitext-103 test	71756544	57%	18.12
GPT-2 small TT rank 64	Wikitext-103 train	Wikitext-103 test	83606784	67%	17.34

Experiments on a GPT model of a medium size

Model	Training	Validation	Number of parameters	Percent of classic model size	Perplexity
GPT-2 medium	Webtext	Wikitext-103 test	354823168	100	21.39
GPT-2 medium TT rank 72	Openwebtext	Wikitext-103 test	218303488	61	31.85
GPT-2 medium SVD rank 50	Openwebtext	Wikitext-103 test	220920832	61	55.1
Distill GPT-2	Openwebtext	Wikitext-103 test	81912576	23	51.45

Kronecker Representation, From Scratch + Distillation

KnGPT2 represent[5] weight matrix $\mathbf{W} \in \mathcal{R}^{m \times n}$ as $\mathbf{W} = \mathbf{A} \otimes \mathbf{B}$, where $\mathbf{A} \in \mathcal{R}^{m_1 \times n_1}$, $\mathbf{B} \in \mathcal{R}^{m_2 \times n_2}$. Compression rate $= \frac{m_1 n_1 + m_2 n_2}{mn}$

Model	CoLA	RTE	MRPC	SST-2	MNLI	QNLI	QQP	Average
GPT-2$_{\text{Small}}$	47.6	69.31	87.47	92.08	83.12	88.87	90.25	79.81
DistilGPT2	38.7	65.0	87.7	91.3	79.9	85.7	89.3	76.8
DistilGPT2 + KD	38.64	64.98	87.31	89.80	80.42	86.36	89.61	76.73
KnGPT2	37.51	**70.4**	**88.55**	88.64	78.93	86.10	88.87	77
KnGPT2 + ILKD	**45.36**	69.67	87.41	**91.28**	**82.15**	**88.58**	**90.34**	**79.25**

Table 5: This table shows performance of the models on dev set of GLUE tasks. Note that GPT-2$_{\text{Small}}$ is used as teacher for KD.

KD means Knowledge Distillation, ILKD means "Intermediate layer Knowledge Distillation"

[5] https://arxiv.org/pdf/2110.08152.pdf

Thank you for your attention =)